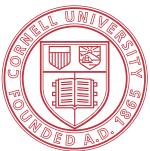


ECE 6775
High-Level Digital Design Automation
Fall 2024

Tutorial on C-Based HLS



Cornell University



Agenda

- ▶ Introduction to high-level synthesis (HLS)
 - C-based synthesis
 - Common HLS optimizations
- ▶ Matrix-vector multiplication using Vivado HLS

High-Level Synthesis (HLS)

► What

- An automated design process that transforms **high-level functional specifications into optimized register-transfer level (RTL)** descriptions for efficient hardware implementation
 - Input spec. to HLS is typically untimed or partially timed

► Why

- **Productivity:** Lower design complexity & faster simulation speed
- **Portability:** Single (untimed) source → multiple implementations
- **Quality:** Quicker design space exploration → higher quality

A Simple Example: RTL vs. HLS

- ▶ A GCD unit with handshake



HLS Code

```

void GCD (
    req_t req,
    resp_t& resp
) {
    short a = req.dat_a;
    short b = req.dat_b;
    while ( a != b ) {
        if ( a > b )
            a = a - b;
        else
            b = b - a;
    }
    resp.dat = a;
}
  
```

Manual RTL (partial)

```

module GcdUnitRTL(
    input wire      clk,
    input wire [31:0] req_dat,
    output wire      req_rdy,
    input wire      req_val,
    input wire      reset,
    output wire [15:0] resp_dat,
    input wire      resp_rdy,
    output wire      resp_val
);
  
```

```

always @(*) begin
    if (curr_state == STATE_IDLE)
        if (req_val)
            next_state = STATE_CALC;
    if (curr_state == STATE_CALC)
        if (!is_a_lt_b && is_b_zero)
            next_state = STATE_DONE;
    if (curr_state == STATE_DONE)
        if (resp_val && resp_rdy)
            next_state = STATE_IDLE;
end
  
```

Module declaration

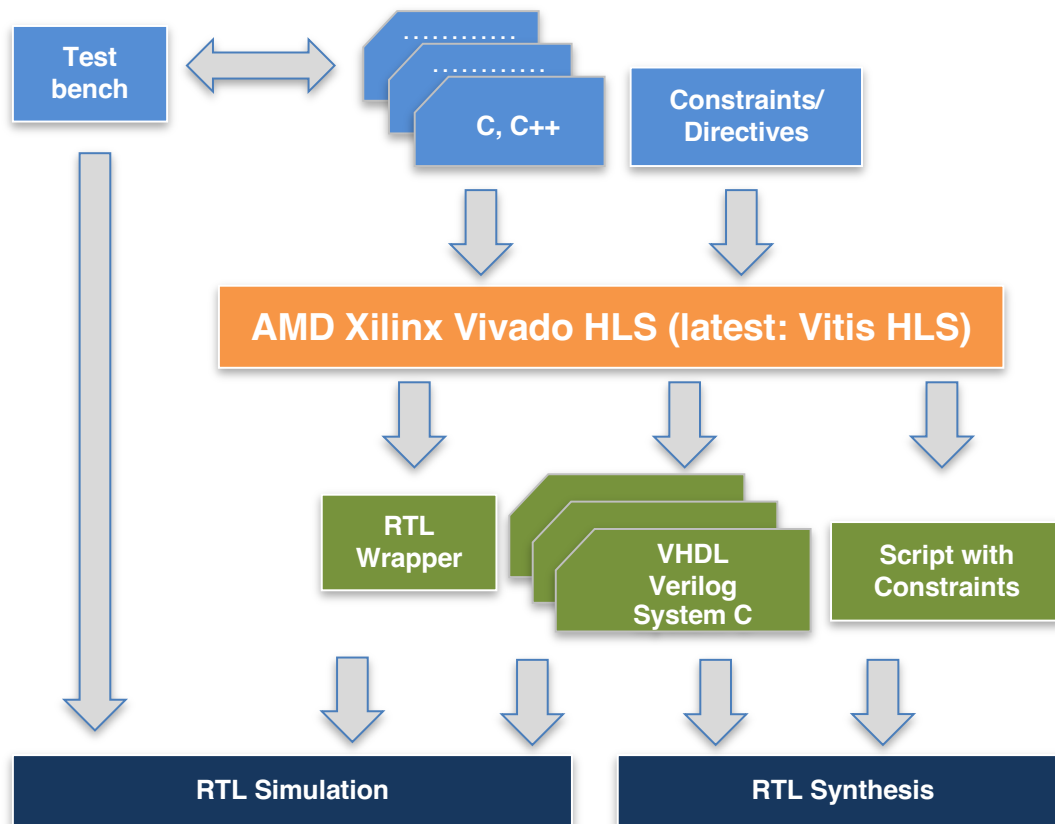
State transition

```

always @ (*) begin
    if (current_state == STATE_IDLE) begin
        req_rdy = 1; resp_val = 0;
        a_mux_sel = A_MUX_SEL_IN;
        b_mux_sel = B_MUX_SEL_IN;
        a_reg_en = 1; b_reg_en = 1;
    end
    if (current_state == STATE_CALC) begin
        req_rdy = 0; resp_val = 0;
        do_swap = is_a_lt_b;
        do_sub = ~is_b_zero;
        a_mux_sel = do_swap ? A_MUX_SEL_B : A_MUX_SEL_SUB;
        a_reg_en = 1; b_reg_en = do_swap;
        b_mux_sel = B_MUX_SEL_A;
    end
    if (current_state == STATE_DONE) begin
        req_rdy = 0; resp_val = 1;
        a_mux_sel = A_MUX_SEL_X;
        b_mux_sel = B_MUX_SEL_X;
        a_reg_en = 0; b_reg_en = 0;
    end
end
  
```

Output logic

A Representative C-Based HLS Tool



**~10X code size reduction with
algorithmic specification**

~100X simulation speedup

Behavioral-level IP reuse

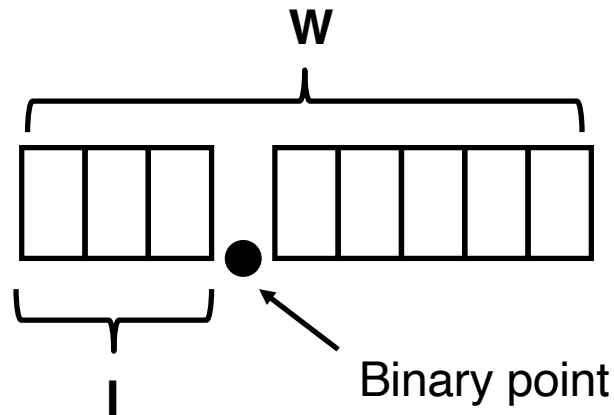
Fast architecture exploration

Typical C/C++ Synthesizable Subset

- ▶ Data types:
 - **Primitive types:** `int`, `unsigned`, `float`, `double`, ...
 - **Arbitrary precision:** `ap_int`, `ap_fixed`, ...
 - **Composite types:** `array`, `struct`, `union`, ...
 - **Templated types:** `template<>`
 - **Statically determinable pointers**
- ▶ No dynamic memory allocations
 - ~~`malloc`, `new`, `std::vector<>`~~
- ▶ No recursive function calls
- ▶ Prints in csim only

Recap: Fixed-Point Types

- ▶ `ap_fixed` is a templated C++ data type used for representing fixed-point numbers
 - Signed: `ap_fixed`; Unsigned: `ap_ufixed`
 - Template parameters `ap_fixed<W, I, Q, O>`
 - W: total bitwidth
 - I: integer bitwidth
 - Q: quantization mode (optional, default is `AP_TRN`)
 - O: overflow mode (optional, default is `AP_WRAP`)



Arbitrary Precision Integer

- ▶ C/C++ only provides a limited set of native integer types
 - Usually: `char` (8b), `short` (16b), `int` (32b), `long` (64b), `long long` (64b)
 - Byte aligned: efficient in processors
- ▶ Arbitrary precision integer in Vivado HLS
 - Signed: `ap_int`; Unsigned `ap_uint`
 - Two's complement representation for signed integer
 - Templated class `ap_int<W>` or `ap_uint<W>`
 - `W` is the user-specified bitwidth

```
#include <ap_int.h>
ap_int<9>      x; // 9-bit
ap_uint<24>    y; // 24-bit unsigned
ap_uint<512>   z; // 512-bit unsigned
```


Typical C/C++ Constructs to RTL Mapping

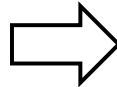
<u>C/C++</u> <u>Constructs</u>		<u>RTL</u> <u>Components</u>
Functions	→	Modules
Arguments	→	Input/output ports
Operators	→	Functional units
Scalars	→	Wires or registers
Arrays	→	Memories
Control flows	→	Control logics

Functions and Design Hierarchy

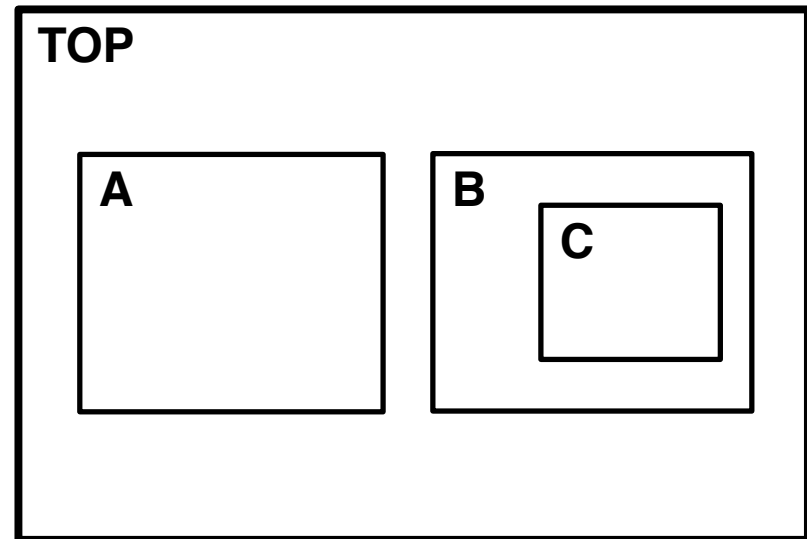
- ▶ Each function is usually translated into an RTL module
 - Function arguments become ports on the RTL blocks
 - Functions may be inlined to dissolve their hierarchy

Source code

```
void A(...) { ... }  
void C(...) { ... }  
void B(...) {  
    C(...);  
    ...  
}  
  
void TOP(...) {  
    A(...); B(...);  
}
```



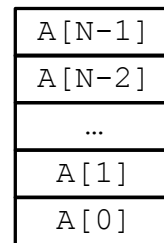
RTL module hierarchy



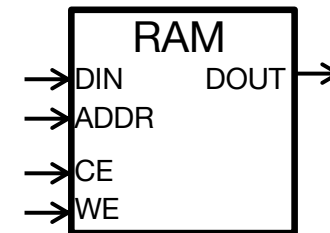
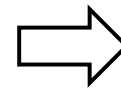
Arrays

- ▶ An array is usually implemented by a memory module in RTL
 - Reading and writing to the array correspond to accessing RAM, while constant arrays are stored in ROM
 - Typically, each memory module supports a limited number of read/write ports, typically up to 2

```
void TOP() {  
    int A[N];  
    for (i = 0; i < N; i++)  
        A[i] = A[i] + i;  
}
```



Array **A[N]**



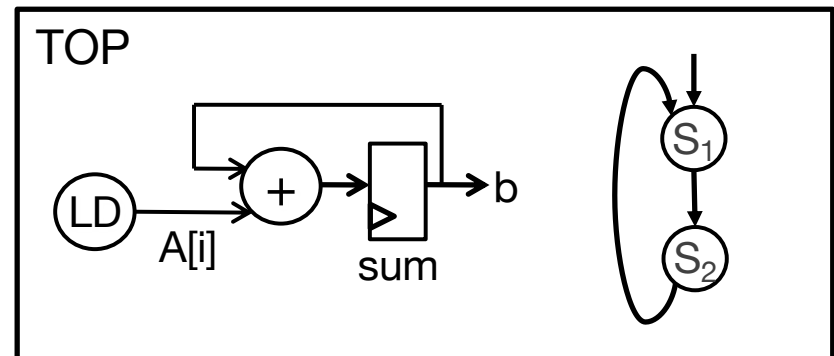
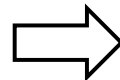
Memory Module(s)

- ▶ An array can be partitioned and implemented with multiple RAMs
 - Extreme case: completely partitioned into individual elements that map to discrete registers
- ▶ Multiples arrays can be merged and mapped to one RAM

Loops

- ▶ By default, loops are “rolled”
 - Each loop iteration corresponds to a “sequence” of states (more generally, an FSM)
 - This state sequence will be repeated multiple times based on the loop trip count (or loop bound)

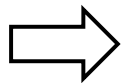
```
void TOP() {  
    ...  
    for (i = 0; i < N; i++)  
        sum += A[i];  
}
```



Loop Unrolling

- Unrolling can expose more parallelism to achieve shorter latency or higher throughput
 - (+) Decreased loop control overhead
 - (+) Increased parallelism for scheduling
 - (-) Increased operation count, which may negatively impact area, timing, and power

```
void TOP() {  
    ...  
    for (i = 0; i < N; i++){  
        #pragma HLS unroll factor=4  
        sum += A[i];  
    }  
}
```

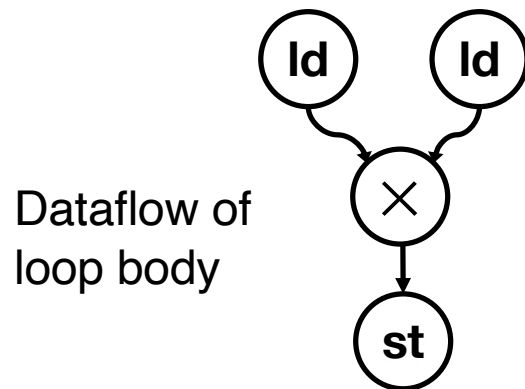


```
void TOP() {  
    ...  
    for (i = 0; i < N/4; i++){  
        sum += A[4*i];  
        sum += A[4*i+1];  
        sum += A[4*i+2];  
        sum += A[4*i+3];  
    }  
}
```

Loop Pipelining

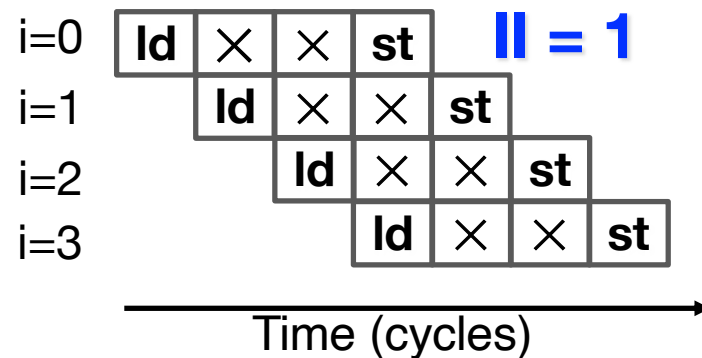
- ▶ Pipelining is one of the most important optimizations for HLS
 - Key factor: **Initiation Interval (II)**
 - Allows a new iteration to begin processing, II cycles after the start of the previous iteration (**II=1 means the loop is fully pipelined**)

```
for (i = 0; i < N; ++i){  
    #pragma HLS pipeline II=1  
    p[i] = x[i] * y[i];  
}
```



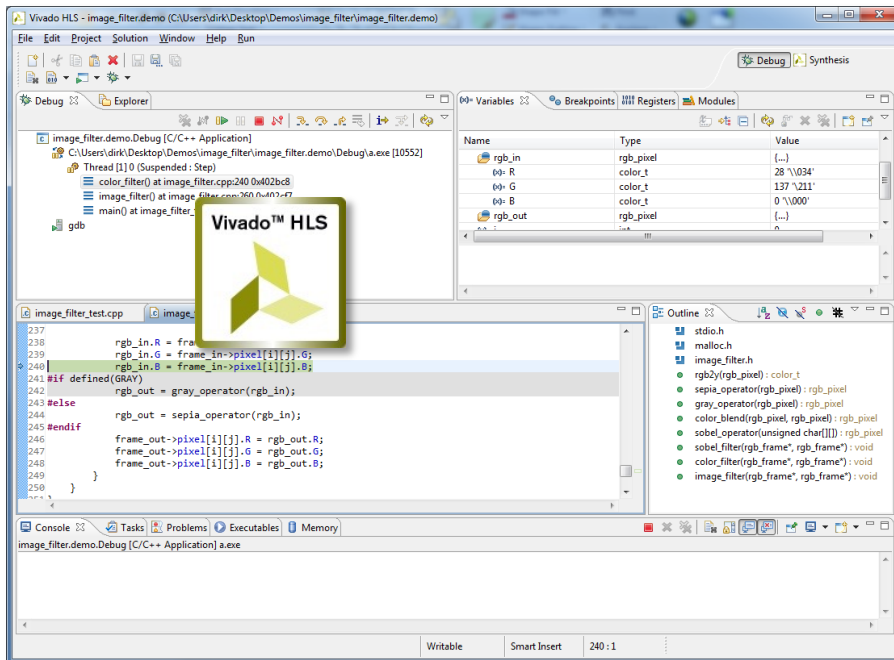
ld – Load (memory read)
st – Store (memory write)

Pipelined schedule



Here we assume multiplication (×) takes two cycles

A Tutorial on Vivado HLS



```
(Vivado19) nz264@brg-zhang-xcel:~/shared/ece5997/mvml-tutorial$ vivado_hls -f run.tcl

***** Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC v2019.2.1 (64-bit)
**** SW Build 2729669 on Thu Dec  5 04:48:12 MST 2019
**** IP Build 2729494 on Thu Dec  5 07:38:25 MST 2019
** Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.

source /opt/xilinx/Xilinx_Vivado_vitis_2019.2/Vivado/2019.2/scripts/vivado_hls/hls.tcl -notrace
INFO: Applying HLS Y2K22 patch v1.2 for IP revision
INFO: [HLS 200-10] Running '/opt/xilinx/Xilinx_Vivado_vitis_2019.2/Vivado/2019.2/bin/unwrapped/lnx64.o/vivado_hls'
INFO: [HLS 200-10] For user 'nz264' on host 'en-ec-brg-zhang-xcel.coecis.cornell.edu' (Linux_x86_64 version 3.10.0-1160.71.1.el7.x86_64) on Mon Aug 22 11:07:33 EDT 2022
INFO: [HLS 200-10] On os "CentOS Linux release 7.9.2009 (Core)"
INFO: [HLS 200-10] In directory '/work/shared/users/phd/nz264/ece5997/mvml-tutorial'
Sourcing Tcl script 'run.tcl'
INFO: [HLS 200-10] Opening and resetting project '/work/shared/users/phd/nz264/ece5997/mvml-tutorial/mvml_vitis.prj'.
INFO: [HLS 200-10] Adding design file 'mvml_unroll.c' to the project
INFO: [HLS 200-10] Adding test bench file 'mvml_top.c' to the project
INFO: [HLS 200-10] Opening and resetting solution '/work/shared/users/phd/nz264/ece5997/mvml-tutorial/mvml_vitis.prj/solution1'.
INFO: [HLS 200-10] Cleaning up the solution database.
INFO: [HLS 200-10] Setting target device to 'xc7z020-clg484-1'
INFO: [SYN 201-201] Setting up clock 'default' with a period of 10ns.
INFO: [SCHD 204-61] Option 'relax_ii_for_timing' is enabled, will increase II to preserve clock frequency constraints.
INFO: [HLS 200-10] Analyzing design file 'mvml_unroll.c' ...
INFO: [HLS 200-111] Finished Linking Time (s): cpu = 00:00:11; elapsed = 00:00:18 . Memory (MB): peak = 1057.715; gain = 527.219; free physical = 97063; free virtual = 219377
INFO: [HLS 200-111] Finished Checking Pragmas Time (s): cpu = 00:00:11; elapsed = 00:00:18 . Memory (MB): peak = 1057.715; gain = 527.219; free physical = 97063; free virtual = 219377
INFO: [HLS 200-10] Starting code transformations ...
INFO: [HLS 200-111] Finished Standard Transforms Time (s): cpu = 00:00:12; elapsed = 00:00:19 . Memory (MB): peak = 1057.715; gain = 527.219; free physical = 97048; free virtual = 219361
INFO: [HLS 200-10] Checking synthesizability ...
INFO: [HLS 200-111] Finished Checking Synthesizability Time (s): cpu = 00:00:12; elapsed = 00:00:19 . Memory (MB): peak = 1057.715; gain = 527.219; free physical = 97063; free virtual = 219377
INFO: [HLS 200-489] Unrolling loop 'ACC_LOOP' (mvml_unroll.c:17) in function 'mvml' completely with a factor of 16.
INFO: [XFORM 203-11] Balancing expressions in function 'mvml' (mvml_unroll.c:6)...15 expression(s) balanced.
INFO: [HLS 200-111] Finished Pre-synthesis Time (s): cpu = 00:00:12; elapsed = 00:00:19 . Memory (MB): peak = 1057.715; gain = 527.219; free physical = 97044; free virtual = 219358
INFO: [HLS 200-111] Finished Architecture Synthesis Time (s): cpu = 00:00:12; elapsed = 00:00:19 . Memory (MB): peak = 1057.715; gain = 527.219; free physical = 97043; free virtual = 219357
INFO: [HLS 200-10] Starting hardware synthesis ...
INFO: [HLS 200-10] Synthesizing 'mvml' ...
WARNING: [SYN 201-107] Renaming port name 'mvml/output' to 'mvml/output_r' to avoid the conflict with HDL keywords or other object names.
INFO: [HLS 200-10] -----
INFO: [HLS 200-42] -- Implementing module 'mvml'
INFO: [HLS 200-10] -----
INFO: [HLS 200-10] Starting scheduling ...
INFO: [SCHD 204-11] Finished scheduling.
```

AMD Xilinx Vivado HLS (v2019.2)
(We will exclusively use the command-line interface)

Matrix-Vector Multiplication

$$\mathbf{y} = \mathbf{A} \mathbf{x} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N \\ \vdots \\ a_{N1}x_1 + a_{N2}x_2 + \cdots + a_{NN}x_N \end{bmatrix}$$
$$y[i] = \sum_{j=0}^N A[i][j] \times x[j]$$

A: input matrix

x : input vector

y : output vector

```
// mv.cpp
// original, non-optimized version of matrix-
// vector multiplication

#include "mv.h" // N is defined 16 here

void MV(int A[N][N], int x[N], int y[N]) {
    int i, j;
    int acc;
    OUTER:
        for (i = 0; i < N; i++) {
            acc = 0;
            INNER:
                for (j = 0; j < N; j++) {
                    acc += A[i][j] * x[j];
                }
            y[i] = acc;
        }
}
```


Activity: Setup on ECE Linux Server

- ▶ Log into ecelinux server

```
> ssh <netid>@ecelinux.ece.cornell.edu
```

- More info: it.coecis.cornell.edu/ece/ecelinux/

- ▶ Get Vivado HLS tool in your environment

- Source class setup script to setup Vivado HLS

```
> source /classes/ece6775/setup-ece6775.sh
```

- ▶ Copy MV Example to Your Working Directory

- Does not have to be `~/ece6775`

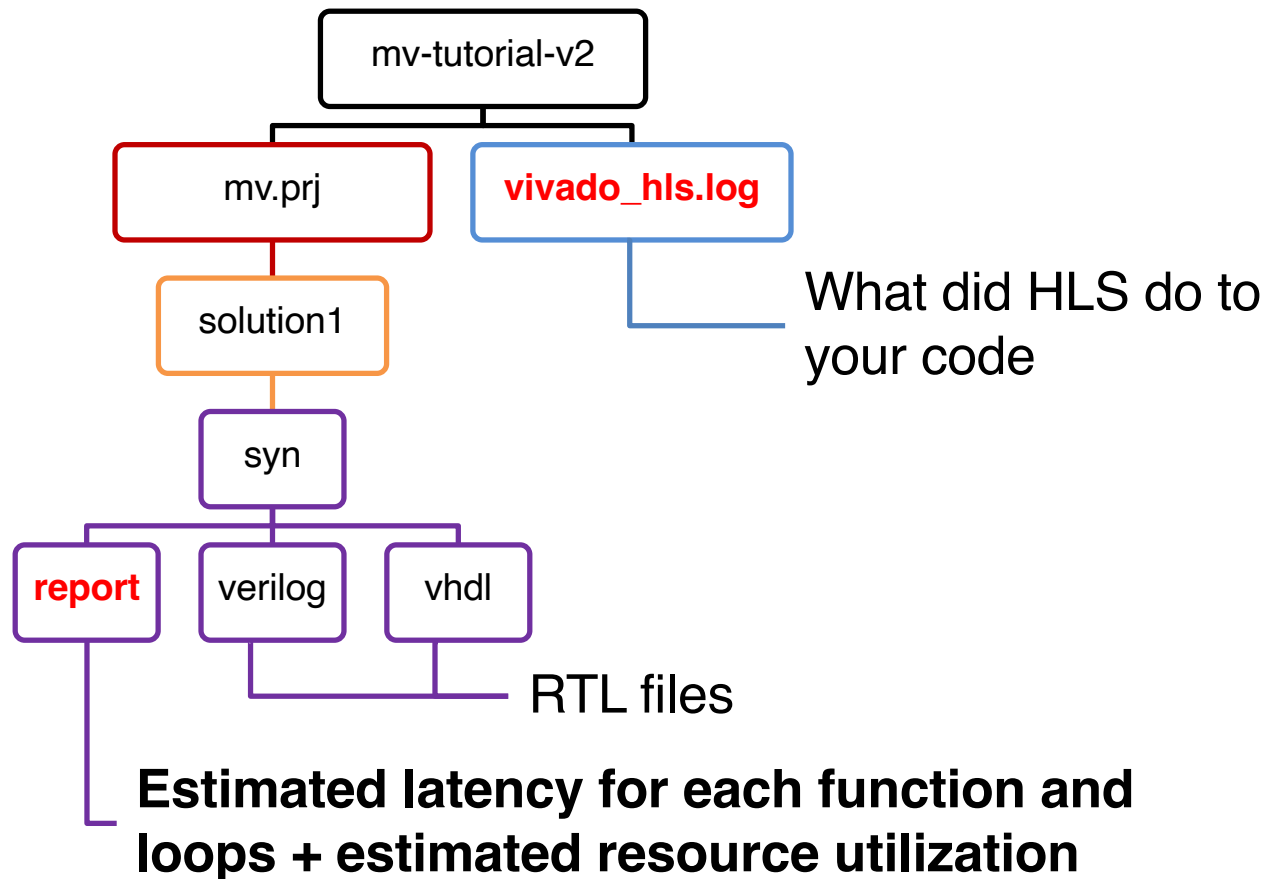
```
> mkdir -p ~/ece6775
> cd ~/ece6775
> cp -r /classes/ece6775/mv-tutorial-v2/ .
> ls
mv.cpp mv.h run.tcl testbench.cpp
```

Goal of this Tutorial

- ▶ Apply different optimizations and observe results
 - Throughput = operation count / top-level function latency
 - Operation count = $16 \times 16 \times 2 = 512$
 - Area = DSP + FF + LUT usage
- ▶ We will fill in this result table after synthesizing each design

Design	Throughput (Ops/Cycle)	Area (DSP+FF+LUT)
baseline		
unroll		
unroll + pipeline		
unroll + partition + pipeline		

Project Directory Structure



Activity: Run the Baseline Design

► Run csynth

– `> vivado_hls -f run.tcl`

► Read the latency of loop OUTER

– See *mv.prj/solution1/syn/report/MV_csynth.rpt*

```
+ Detail:
* Instance:
N/A

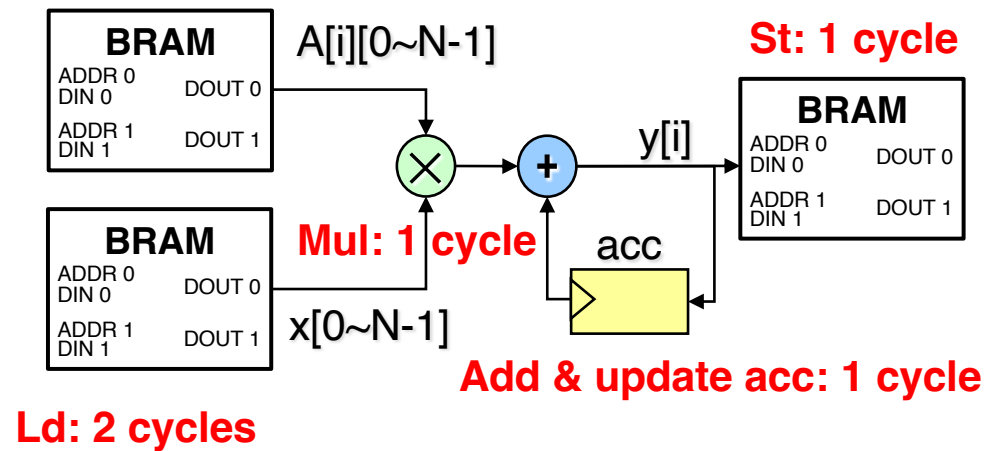
* Loop:
```

		Latency (cycles)		Iteration	Initiation Interval		Trip	
Loop Name	min	max	Latency	achieved	target	Count	Pipelined	
- OUTER	16	16	16	-	-	16	no	
+ INNER	16	16	16	-	-	16	no	

The Baseline Micro-architecture

- Latency of OUTER should read 1056

```
// N = 16
OUTER:
  for (i = 0; i < N; i++) {
    acc = 0;
    INNER:
      for (j = 0; j < N; j++) {
        acc += A[i][j] * x[j];
      }
      y[i] = acc;
  }
```

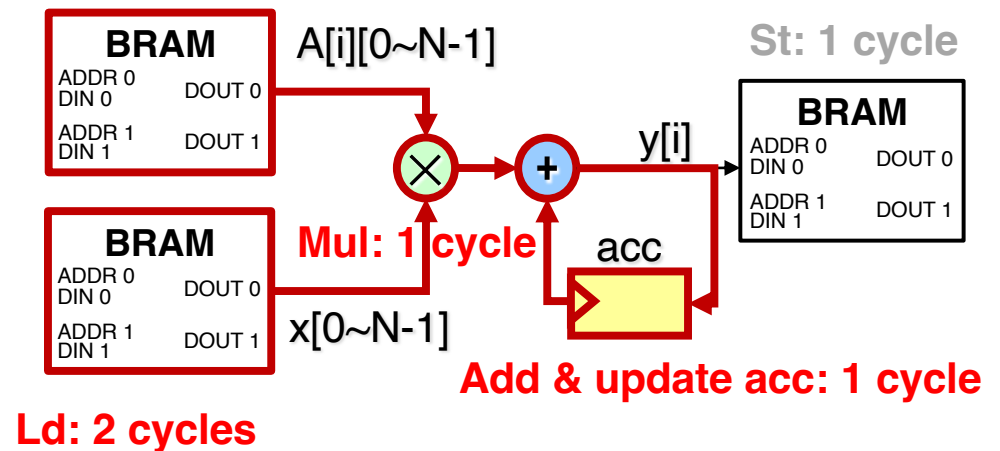


- Latency of INNER =

The Baseline Micro-architecture

- Latency of OUTER should read 1056

```
// N = 16
OUTER:
  for (i = 0; i < N; i++) {
    acc = 0;
    INNER:
      for (j = 0; j < N; j++) {
        acc += A[i][j] * x[j];
      }
    y[i] = acc;
  }
```

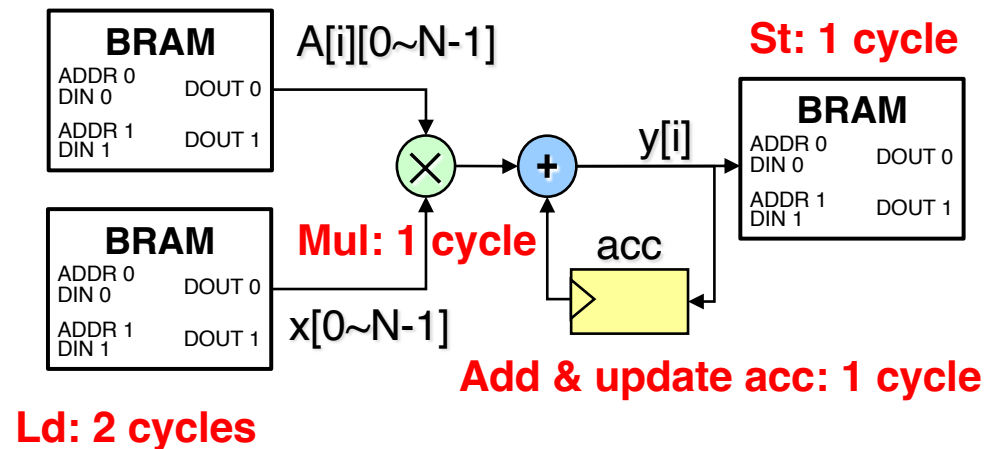


- Latency of INNER = $(2+1+1) \times 16 = 64$

The Baseline Micro-architecture

- Latency of OUTER should read 1056

```
// N = 16
OUTER:
  for (i = 0; i < N; i++) {
    acc = 0;
    INNER:
      for (j = 0; j < N; j++) {
        acc += A[i][j] * x[j];
      }
      y[i] = acc;
  }
```



- Latency of INNER = $(2+1+1) \times 16 = 64$
- Latency of OUTER = $(64+2) \times 16 = 1056$

Result Table

- ▶ Throughput = operation count / top-level function latency
 - Operation count = $16 \times 16 \times 2 = 512$
- ▶ Area = DSP + FF + LUT usage

Design	Throughput (Ops/Cycle)	Area (DSP+FF+LUT)
baseline	0.484	357
unroll		
unroll + pipeline		
unroll + partition + pipeline		

Activity: Unroll Inner Loop

- ▶ Add the unroll pragma to loop INNER

```
- // N = 16
  OUTER:
    for (i = 0; i < N; i++) {
      acc = 0;
      INNER:
        for (j = 0; j < N; j++) {
          #pragma HLS unroll
          acc += A[i][j] * x[j];
        }
      y[i] = acc;
    }
```

- ▶ Run csynth

```
- > vivado_hls -f run.tcl
```

- You may use a different project name to keep the baseline design

Result Table

- ▶ Throughput = operation count / top-level function latency
 - Operation count = $16 \times 16 \times 2 = 512$
- ▶ Area = DSP + FF + LUT usage

Design	Throughput (Ops/Cycle)	Area (DSP+FF+LUT)
baseline	0.484	357
unroll	2.653	1030
unroll + pipeline		
unroll + partition + pipeline		

Activity: Pipeline Outer Loop

- ▶ Add the pipeline pragma to loop OUTER

```
- // N = 16
  OUTER:
    for (i = 0; i < N; i++) {
      #pragma HLS pipeline II=1
      acc = 0;
      INNER:
        for (j = 0; j < N; j++) {
          #pragma HLS unroll
          acc += A[i][j] * x[j];
        }
      y[i] = acc;
    }
```

The inner loop will be automatically unrolled due to pipeline, but we still keep the pragma for clarity.

- ▶ What is the final II of the loop?
 - Hint: how many elements to load per cycle?
- ▶ Run csynth and check the log

```
- > vivado_hls -f run.tcl
```

The Reason of Pipeline Failure

- ▶ With an II = 1, we expect to load 16 elements of A and x every cycle
- ▶ But one BRAM only has 2 ports
- ▶ The II is relaxed to $16 / 2 = 8$

```
INFO: [SCHED 204-61] Pipelining loop 'OUTER'.  
WARNING: [SCHED 204-69] Unable to schedule 'load'  
operation ('A_load_2', mv.cpp:16) on array 'A' due  
to limited memory ports. Please consider using a  
memory core with more ports or partitioning the  
array 'A'.  
INFO: [SCHED 204-61] Pipelining result : Target II =  
1, Final II = 8, Depth = 12.
```

Result Table

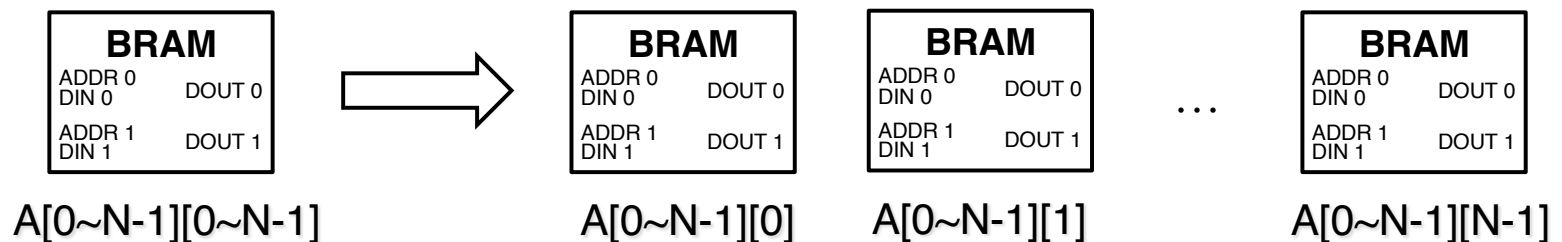
- ▶ Throughput = operation count / top-level function latency
 - Operation count = $16 \times 16 \times 2 = 512$
- ▶ Area = DSP + FF + LUT usage

Design	Throughput (Ops/Cycle)	Area (DSP+FF+LUT)
baseline	0.484	357
unroll	2.653	1030
unroll + pipeline	3.850	1047
unroll + partition + pipeline		

The Solution: Partition Arrays

```
INFO: [SCHD 204-61] Pipelining loop 'OUTER'.  
WARNING: [SCHD 204-69] Unable to schedule 'load'  
operation ('A_load_2', mv.cpp:16) on array 'A' due  
to limited memory ports. Please consider using a  
memory core with more ports or partitioning the  
array 'A'.  
INFO: [SCHD 204-61] Pipelining result : Target II =  
1, Final II = 8, Depth = 12.
```

- ▶ Array partitioning breaks one array into smaller portions and implements it with multiple BRAM modules



Activity: Partition Arrays

- Add the following two pragmas in function MV:

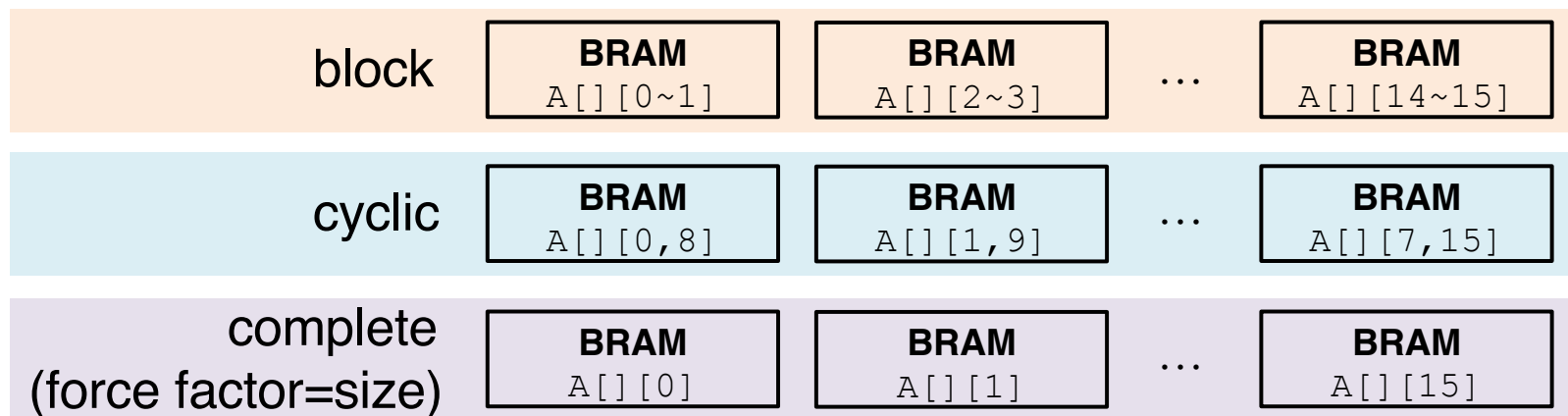
```
void MV(int A[N][N], int x[N], int y[N]) {  
    #pragma HLS array_partition block variable=A dim=2 factor=8  
    #pragma HLS array_partition block variable=x factor=8  
    int i, j;  
    int acc;  
    OUTER:  
    for (i = 0; i < N; i++) {  
        #pragma HLS pipeline II=1  
        acc = 0;  
        INNER:  
        for (j = 0; j < N; j++) {  
            #pragma HLS unroll  
            acc += A[i][j] * x[j];  
        }  
        y[i] = acc;  
    }  
}
```

- Run csynth

```
- > vivado_hls -f run.tcl
```

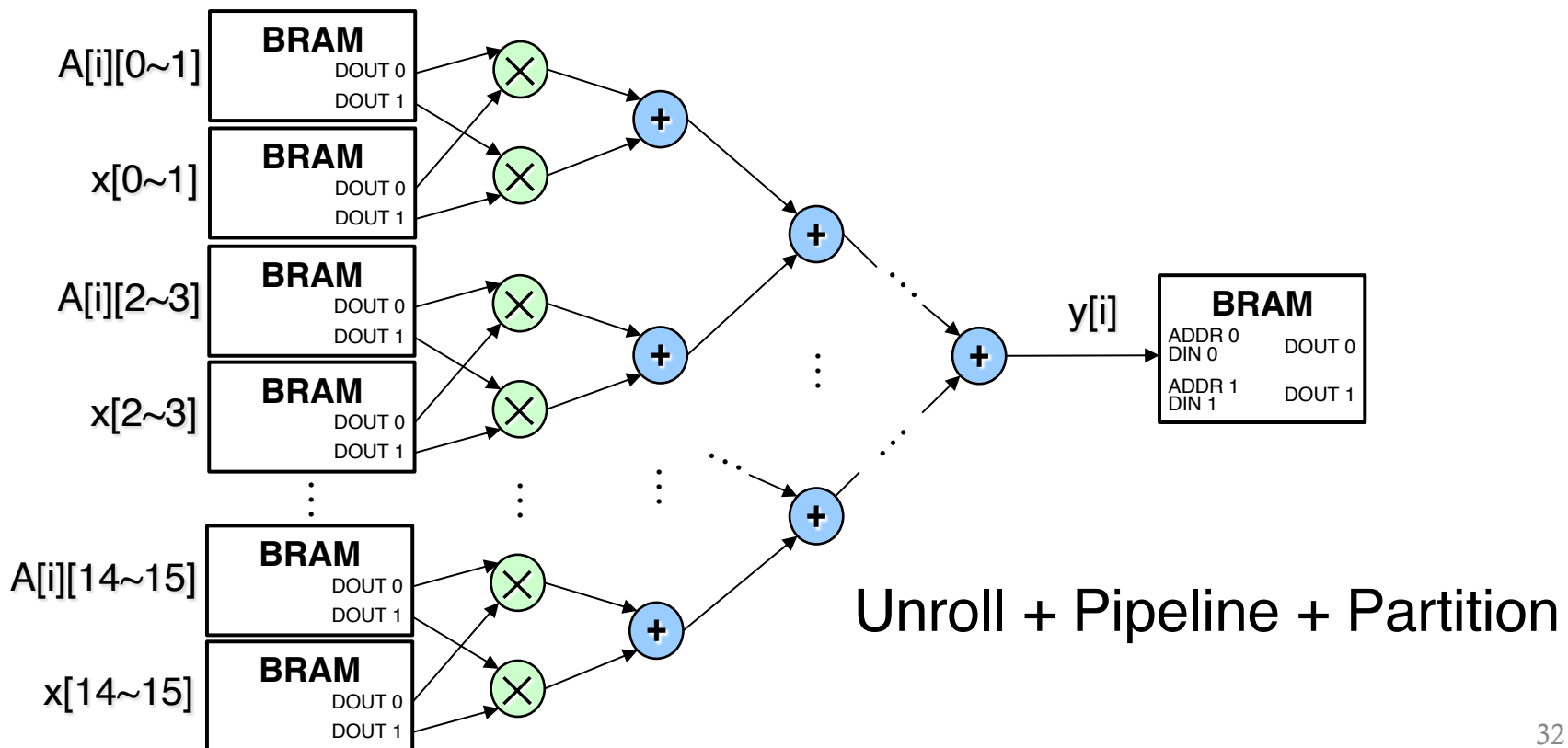
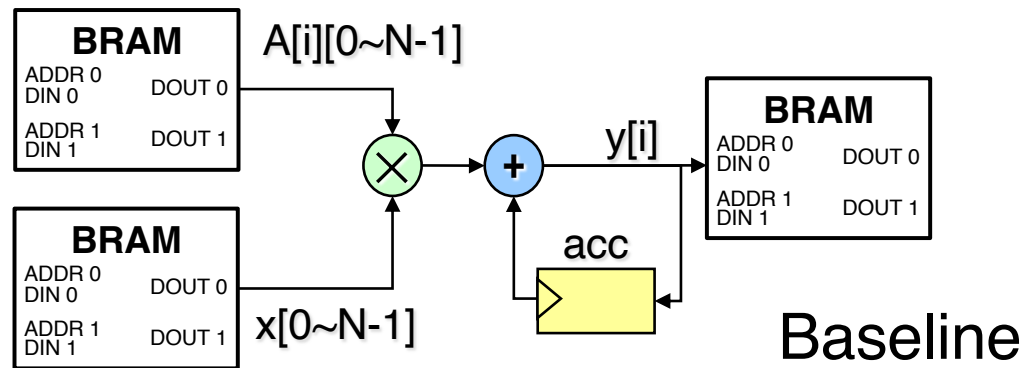
Activity: Partition Arrays

- ▶ `#pragma HLS array_partition block`
`variable=A dim=2 factor=8`
 - **dim**: dimension to be partitioned, select according to the unrolled loop (dim=0 partitions all dimensions)
 - **factor**: number of small arrays after partition
 - **block**: mode of partition (other modes are cyclic and complete)

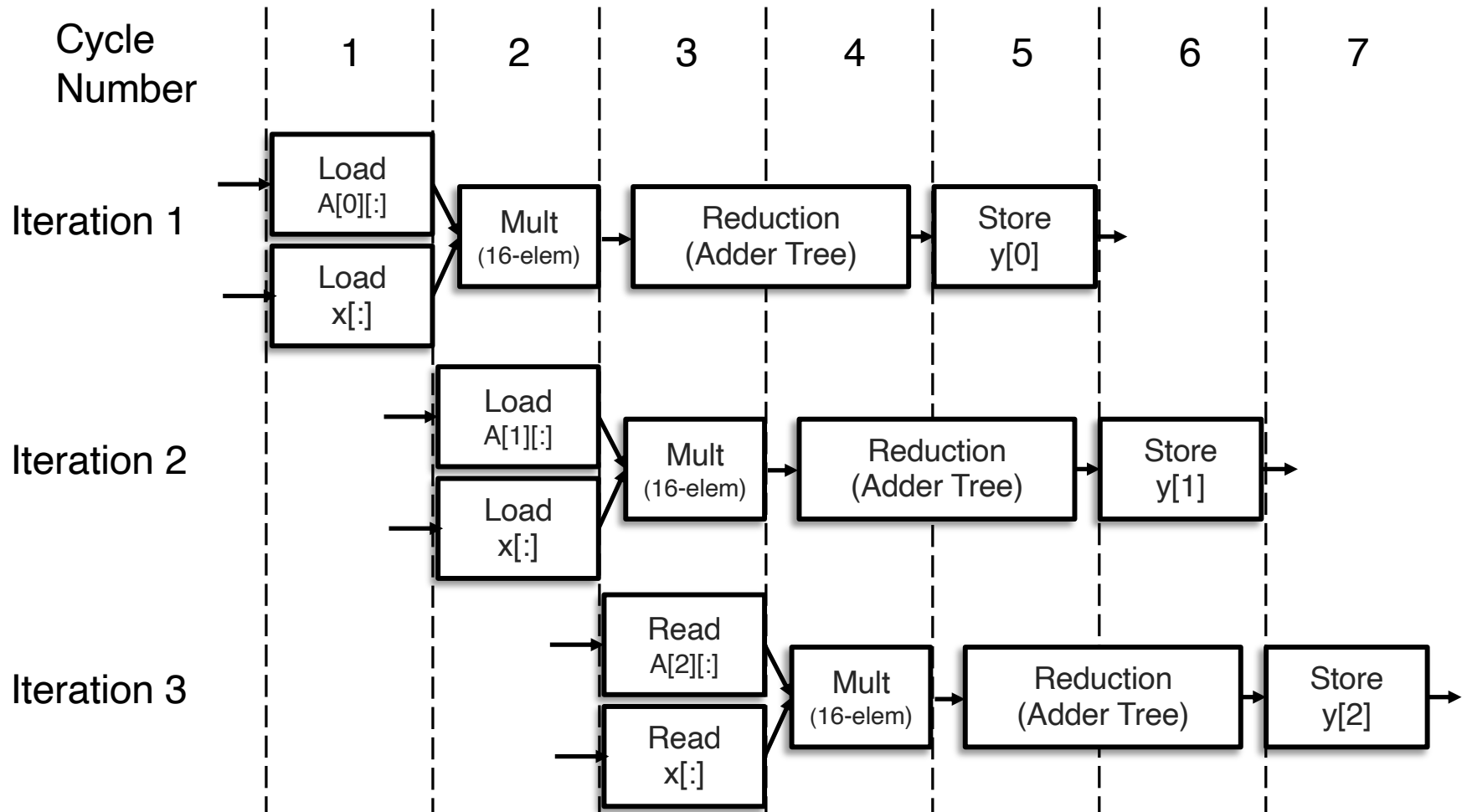


- ▶ In our case, both block and cyclic will achieve $II = 1$
- ▶ When all dimensions are completely partitioned, array will be implemented with registers

Microarchitecture of the Optimized Design



Pipeline Schedule of the Optimized Design



Result Table

- ▶ Throughput = operation count / top-level function latency
 - Operation count = $16 \times 16 \times 2 = 512$
- ▶ Area = DSP + FF + LUT usage

Design	Throughput (Ops/Cycle)	Area (DSP+FF+LUT)
baseline	0.484	357
unroll	2.653	1030
unroll + pipeline	3.850	1047
unroll + partition + pipeline	23.27	2862

48x speedup

8x area

Next Lecture

- ▶ Analysis of Algorithms